Java: Basilari di Programmazione in Piccolo

Sommario: 19 Aprile, 2018

- Java Language Specification: Java SE 8 Edition
- Variabili, Fields e Scope degli Identificatori
- Tipi Primitivi
- Espressioni, Statements, Blocchi
- Importanti default di Java: Object, 0-arity Constructor, this, super.
- Rappresentazione Interna di Programma: Tabella delle Classi, Classi e AR dei Metodi
- Invocazione di Metodo: Evoluzione dello Stack di Controllo, Memoria Statica e Heap.

Java Language Specification

- **Definizione Ufficiale**
  - Java Language Specification: Java SE 8 Edition
  - 15 febbraio 2015, pagg. 768
  - acquistabile (Amazon, Prentice Hall)
  - scaricabile da:
    - https://docs.oracle.com/javase/specs/jls/se7/html/
    - pagine del corso, sezione materiale.
- **Contiene**:
  - La definizione di ogni costrutto: Sintassi, Comportamento, motivazione e uso
  - La sintassi è fornita in una grammatica completa, riportata nel cap. 19.
- **Consultare**: in caso di dubbi

Fields e Scope

- **Variabili di Istanza (Non-Static Fields)**:
  - Definiscono lo stato degli oggetti della classe
  - Scope: Intera definizione di classe
- **Variabili di Classe (Static Fields)**:
  - Definiscono variabili statiche (degli oggetti) della classe
  - Scope: Intera definizione di classe

```
import java.io.*;
import java.util.*;

public class Ex {
    static int A;
    int A; //-- errore: A è gia definito
    int B;
 //metodi
public void met1 (int x) {
    int x; //-- errore: x è gia definito
    int y;
    y=A+x;
    {
        int y; //-- errore y è gia definito;
        int z = y+B;
    }
}
public int met2 (int A, int B) {
    met1(A+B+this.B);
    Ex.A++;
    return A+B;
}
}
```

Variabili, Parametri e Scope

- **Variabili locali**:
  - Definiscono variabili locali di un metodo o di blocco in-line
  - Scope: Intera definizione di metodo o blocco, rispettivamente
- **Parametri**:
  - Definiscono parametri di un metodo
  - Scope: Intera definizione di metodo

```
import java.io.*;
import java.util.*;

public class Ex {
    static int A;
    int A; //-- errore: A è gia definito
    int B;
 //metodi
public void met1 (int x) {
    int x; //-- errore: x è gia definito
    int y;
    y=A+x;
    {
        int y; //-- errore y è gia definito;
        int z = y+B;
    }
}
public int met2 (int A, int B) {
    met1(A+B+this.B);
    Ex.A++;
    return A+B;
}
}
```

Scope degli identificatori e Ricerca del binding

- In Java tutti gli identificatori hanno scope statico:
  - Ma non è richiesto un Meccanismo per la gestione
- Sono impedite collisioni variabili locali
  - No variable shadowing nei metodi.
- Conflitti tra field e variabili locali o parametri risolti
  - Uso di this: `A+B+this.B`
- Gli Activation Record non richiedono Catena Statica.

- Come troviamo il binding di un identificatore `A`?
  Sia AR il corrente activation record della valutazione del codice in cui occorre A. Cerco nell'ordine, nei seguenti Frame:
  - Frame di AR
  - Frame dell'oggetto, se AR è metodo di istanza,
  - Frame della classe.

Tipi Primitivi

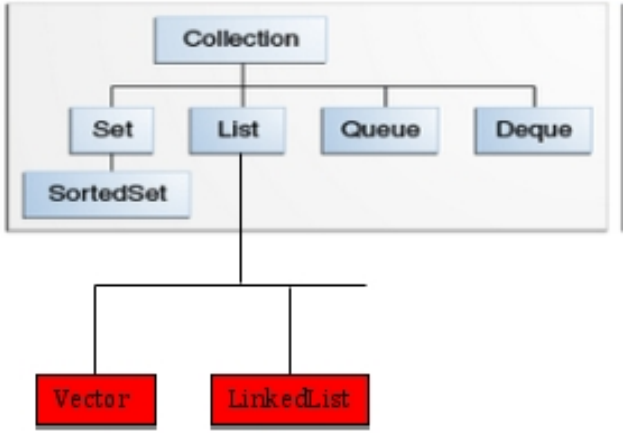
- Scalari: Sono implementati nella macchina ospite

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

- Usabili con gli operatori e nelle forme usuali
- Sono visti come (e convertiti in) oggetti (se necessario)

Tipi Strutturati

- Statici: Array del C/C^{++}
- Dinamici: Di uso comune le sottoclassi dell'interfaccia Collection<T>



- Nei nostri esercizi useremo:
  - Vector<T>: con accesso diretto
  - LinkedList<T>: con accesso sequenziale
- Facili definirne di nuovi
  (in specifiche classi, librerie .jar, API delle proprie applicazioni)

Espressioni, Statements, Blocchi

- Espressioni: Quelle di C/C^{++} sui tipi primitivi e Array Statici
  + Quelle sui nuovi Tipi (classi) + **new** + **invocazione**

- Statements e Blocchi: include la struttura C/C^{++} a parte:
  - identificatori di variabili
  - alcuni costrutti di controllo (iteratori di collezione, gestione eccezioni, synchronized block e concorrenza)
  - sistema dei tipi (sottotipi, e polimorfismo)

Example

Un programma C su valori primitivi e array (e senza statement goto), può essere racchiuso in una classe dove le dichiarazione delle variabili si mantengono (o sono rinominate in caso di collisione), quelle di procedura diventano metodi statici, le invocazioni di procedura diventano invocazioni di metodi, le espressioni si mantengono.

Example

Un programma C su valori primitivi e array (e senza statement goto), può essere incapsulato in una classe dove le dichiarazione delle variabili si mantengono (o sono rinominate in caso di collisione), quelle di procedura diventano metodi statici, le invocazioni di procedura diventano invocazioni di metodi, le espressioni si mantengono.

```
import java.io.*;
import java.util.*;

public class stuck {
    static int taxCalculation(int x){
        return x;}
    public static void main(String [] argv){
        char a = 'e';
        System.out.println("Totale da pagare in euro: e="+a+'='+taxCalculation(10));
    }
}

/* --- Banale rifrasamento in Java del programma C
 #include <stdio.h>
 #include <stdlib.h>

 int taxCalculation(int x){
    return x;}
 int main (int argc, char *argv[]){
    char a = 'e';
    printf("Totale da pagare in euro: e=%2d\n", a+'='+taxCalculation(10));
    return 0;
 } // stampa: Totale da pagare in euro: e=172

 --- Ma ora ha un comportamento corretto
 Marco-Bellias-MacBook-Pro:stuck marcob$ javac stuck.java
 Marco-Bellias-MacBook-Pro:stuck marcob$ java stuck
 Totale da pagare in euro: e=e=10
 Marco-Bellias-MacBook-Pro:stuck marcob$
 }
 */
```

Rappresentazione Interna del Programma: Tabella delle Classi, Classi, Oggetti, Metodi

Una volta analizzato, un programma perde la sua struttura esterna per mostrare la struttura delle classi definite. Ciò è ottenuto dalla seguente struttura che è residente in Memoria Statica e Dinamica per l'intera durata dell'esecuzione del programma sulla JVM

- **Tabella delle Classi**. Coppie (Nome, Descrittore);
- **Descrittore di Classe**. Contiene: Accesso a Superclasse, ...;
- **Oggetto**. Contiene: Accesso a Super Oggetto, a Classe, ...;
- **Descrittore di Metodo**. Contiene: Codice per la creazione...;
- **AR**. Controllo Esecuzione. Contiene: Istanza Template ...

Rappresentazione Interna:Tabella delle Classi

Tabella delle Classi. Coppie (Nome, Descrittore) di ogni classe definita e/o usata dal programma.

- Risiede in Memoria Statica (generata dal Compilatore)
- Il Descrittore è l'accesso alle entità statiche della classe;
- Object è super-Classe quando **extends** è omessa [1].
- Object è la super-Classe al top della gerarchia **extends** di ogni Programma

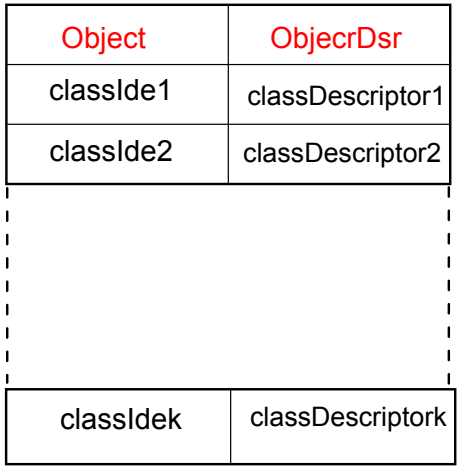


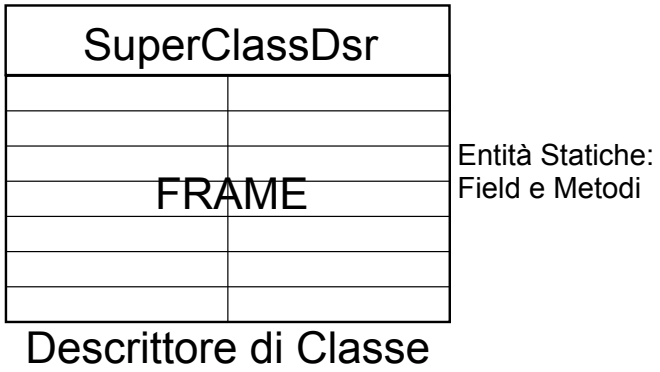
Tabella Classi (Interfacce)

[1] class stuck sopra, nei lucidi, omette la relazione extends

Rappresentazione Interna: Descrittore di Classi

Descrittore di Classe. Contiene: Accesso alla propria Superclasse e il Frame con i Field (statici), i Costruttori di oggetti e i Metodi statici della classe;

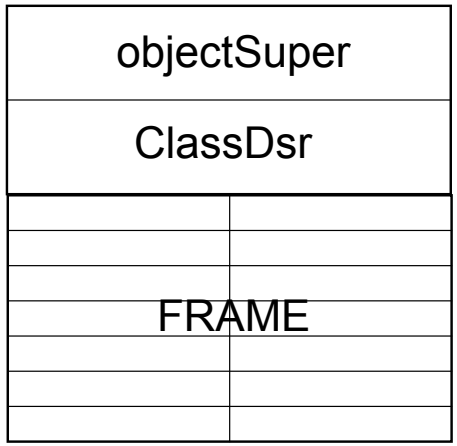
- Risiede in Memoria Statica (generato dal Compilatore)
- I Fields statici sono inizializzati al valore di default del tipo dichiarato per ogni Field.
- Sempre definito un Costruttore di default, di arità 0 che inizializza ogni Field (dinamico) ai valori di default del tipo.



Rappresentazione Interna: Oggetto

Oggetto. Contiene: Accesso al proprio **super** Oggetto, alla propria Classe, al Frame di Field e Metodi di istanza (i.e. dinamici)

- Risiede in Memoria Dinamica (Heap, generato a Run-Time)
- Un **super**-Oggetto esiste sempre. Quando **extends** è omessa, il **super** è l'oggetto della classe **Object**.
- Fields non dichiarati **final** hanno come denotazione un valore modificabile[2]

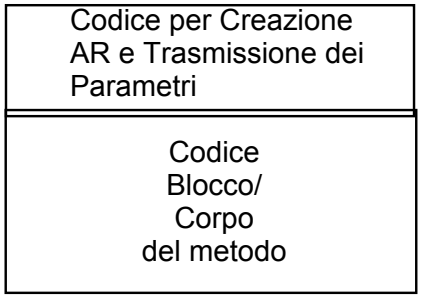


Heap: Struttura di un oggetto

Rappresentazione Interna: Descrittore di Metodo

Descrittore di Metodo. Contiene: Codice per la creazione dello AR e Codice del blocco formante il corpo del Metodo.

- Risiede in Memoria Statica (generato dal Compilatore);
- Il Codice per AR è eseguito all invocazione del Metodo e provvede anche alla trasmissione dei parametri;
- La **trasmissione è solo per valore** e lega formali ad attuali; [3]
- L'AR creato è posto nel top dello Stack di Controllo,



Descrittore di Metodo

[3] vedi trasmissione dei parametri per valore: il formale è legato ad un modificabile inizializzato con il valore dell'attuale

Rappresentazione Interna: Activation Record

AR. Contiene i componenti necessari per la corretta esecuzione del corpo di un Metodo: CD, CLASS/THIS, (RA), CODE, FRAME, RI.

AR : Activation Record

Activation Record: I Componenti

AR. Componenti simili o identici a quelli nei Linguaggi Procedurali

- **CD**. Accesso all'interno dell'AR, posto nel top-1 dello Stack, a cui ritornare il controllo.
- **CLASS/THIS**. Nome di Classe per Metodi Statici, Oggetto altrimenti. Fornisce l'accesso ai Fields (locali e non, statici e dinamici) e ai Metodi (locali o non-locali ereditati);
- **RA**. Indirizzo (nello RI) dell'AR chiamante dove scrivere il valore calcolato;
- **CODE**. Indirizzo dello statement da eseguire (all'interno del corpo del Metodo)
- **FRAME**. Bindings di parametri e locali ottenuti da trasmissione e dichiarazione locali.
- **RI**. Area dei valori intermedi della valutazione di espressioni del corpo del Metodo.

Applichiamo alla classe CounterM

- **Generalità.** Supporremo che CounterM sia una classe di un Programma più ampio.
- **Vediamo prima** le Strutture Interne di un Programma contenente CounterM: Tabella delle Classi, Descrittori di CounterM e dei suoi Metodi;
- **Vediamo poi** un'esecuzione a partire da uno stato *qualunque* che coinvolga codice di CounterM

```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```

Importanti default di Java

- **Object**. Tutte le classi (eccetto la classe Object) hanno una superclasse.

  `... class CounterM {...`

  sta per:

  `... class CounterM extends Object {...`

- **constructors**. Ogni costruttore ha come prima operazione la costruzione dell'oggetto della superclasse.

  Sia `A` il nome di una classe, e

  `A(p1,...,pk){s1;...;sn;}` la dichiarazione di un k-arity constructor

  Allora se s1 è diverso da `super()`, il costruttore sta per:

  `A(p1,...,pk){super();s1;...;sn;}`

- **0-arity constructor**. Tutte le classi hanno un costruttore di arità 0.

  Sia `A` il nome di una classe priva di un tale costruttore.

  Allora tale classe è considerata estesa con il costruttore 0-arity:

  `A(){super();}`

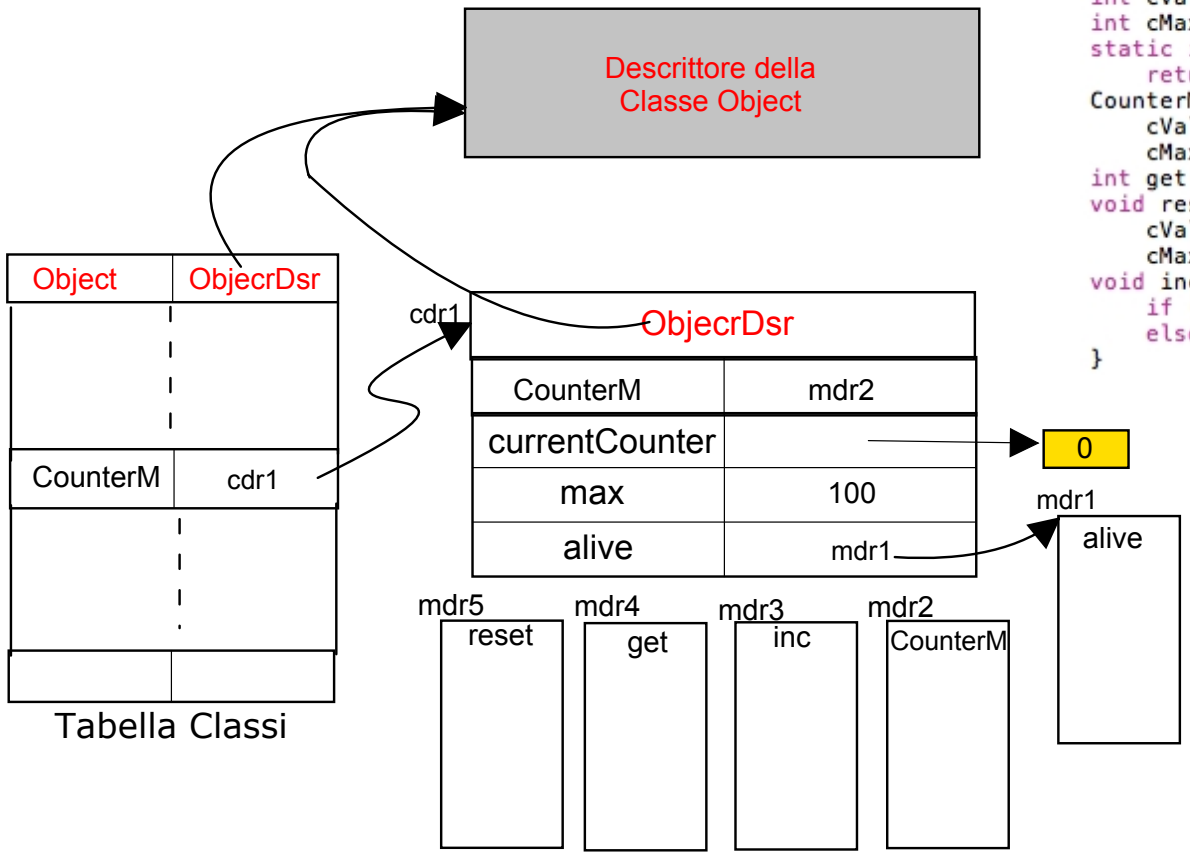
- **this**. L'uso dell'identificatore di self reference this può essere omesso nella selezione delle entitá di un oggetto.

  `cValue, cMax, reset(resetValue)`

  usati nella classe CounterM, della slide precedente, stanno per:

  `this.cValue, this.cMax, this.reset(resetValue)`

Le Strutture Interne: Programma contenente CounterM

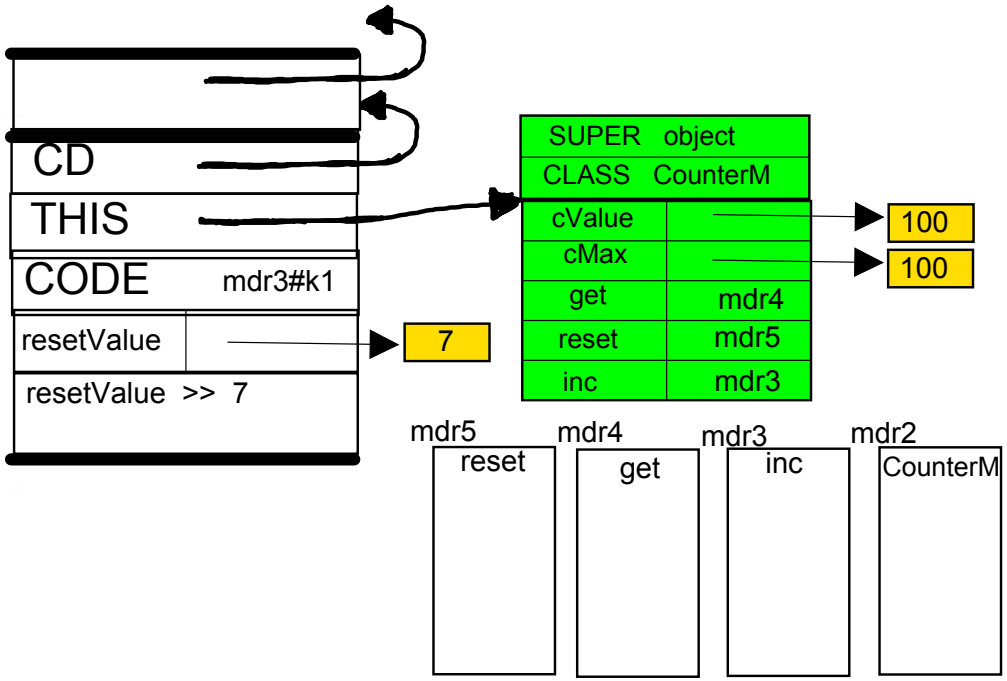



```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```

Applichiamo ad un esecuzione con codice di CounterM

- A destra un possibile stato di Stack di Controllo e Memoria.
- AR ha CODE che indirizza al codice del Metodo `inc`
- Supponiamo che `k1` sia l'indirizzo di `reset(resetValue)`.
- Notare l'oggetto di tipo CounterM, nello heap.

```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```



Invocazione-Applicazione di Metodo (di istanza)

Richiede un AR analogamente all'invocazione di procedura ricorsiva. Ma è espressa nella forma:

$$\mathbf{E_0}.\mathbf{Name}(\mathbf{E_1}, ..., \mathbf{En})$$

dove:

- $\mathbf{E_0}$: Espressione che calcola l'oggetto a cui applicare il metodo;
- **Name**: Nome di un metodo dell'oggetto $\mathbf{E_0}$
- $\mathbf{E_i}$ Espressioni che calcolano gli $\mathtt{n}$ argomenti dell'invocazione;

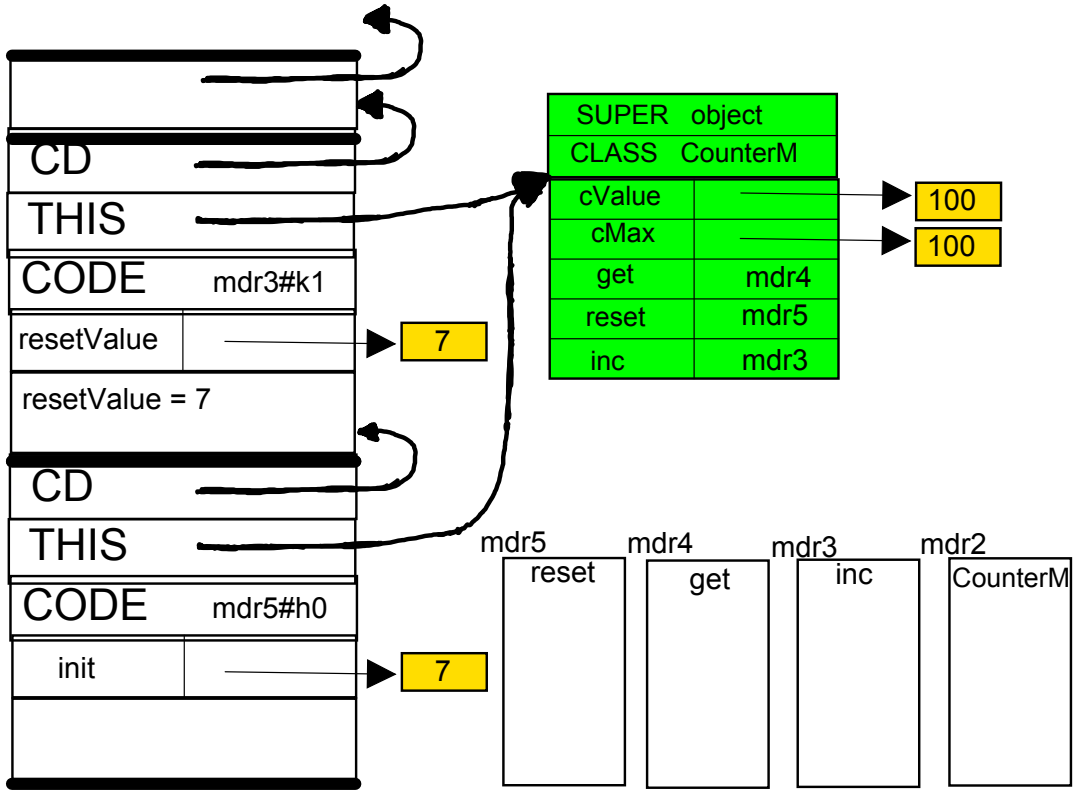
- La trasmissione degli argomenti è sempre e solo per valore.
- Il parametro attuale è stato calcolato (vedi RI) e vale 7.
- lo AR è creato dal codice nel descrittore di reset

- Otteniamo il nuovo stato con il seguente Stack di Controllo

Invocazione di Metodo (di istanza)/2

- A destra lo stato dello Stack di Controllo e Memoria durante l'invocazione del metodo `reset`.
- Supporremo che `h0` indirizzi il codice all'inizio dl corpo.

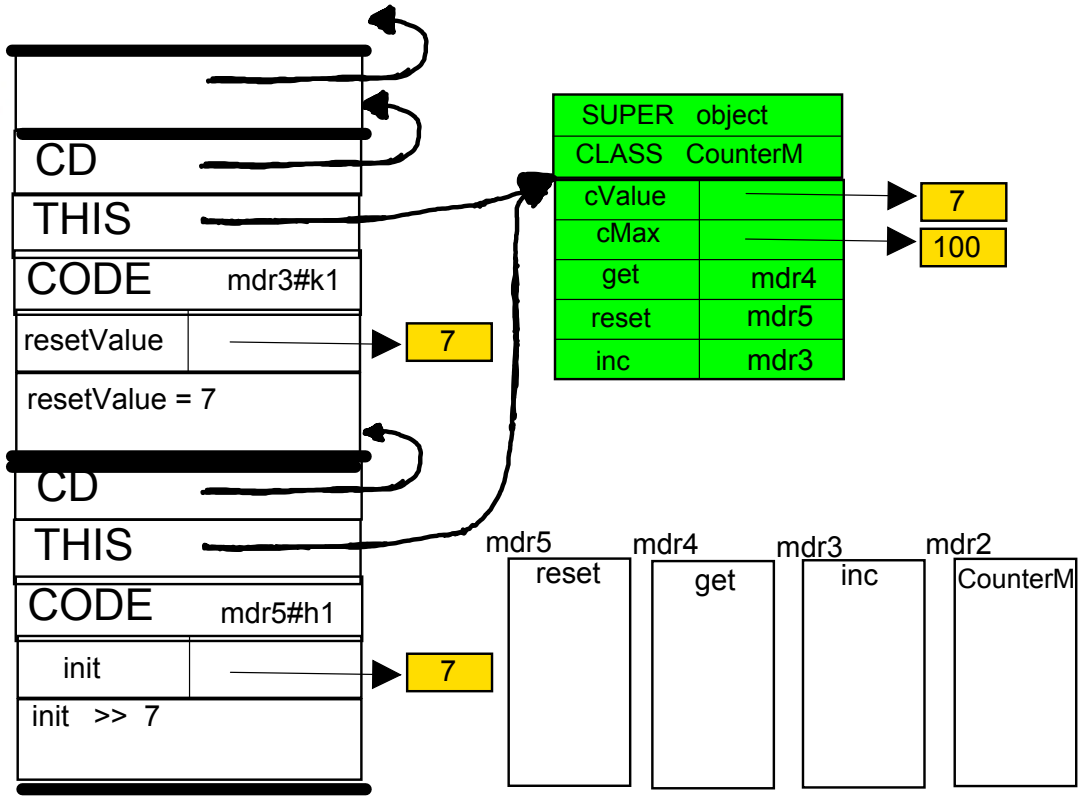
```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```



Invocazione di Metodo (di istanza)/3

- A destra lo stato dello Stack di Controllo e della Memoria durante l'invocazione del metodo `reset`..
- `h1` indirizza il codice successivo al primo assegnamento.

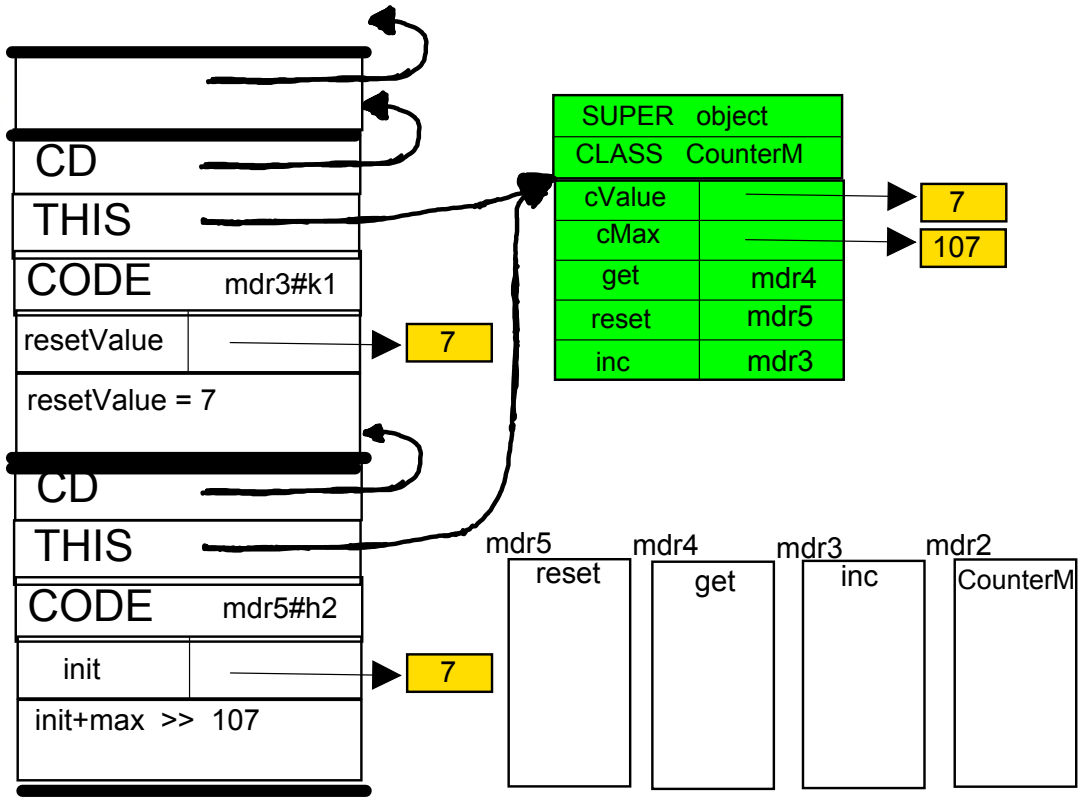
```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```



Invocazione di Metodo (di istanza)/4

- A destra lo stato dello Stack di Controllo e della Memoria durante l'invocazione del metodo `reset`..
- `h2` indirizza il codice successivo al secondo assegnamento.

```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```



Invocazione di Metodo (di istanza)/5

- A destra lo stato dello Stack di Controllo e della Memoria al termine dell'invocazione.

```
public class CounterM {
static int currentCounter = 0;
static final int max = 100;
int cValue;
int cMax;
static int alive () {
    return currentCounter;}
CounterM(int init){
    cValue = init;
    cMax = init+max;}
int get(){return cValue;}
void reset(int init){
    cValue = init;
    cMax = init+max;}
void inc (int resetValue){
    if (cMax>cValue) cValue++;
    else reset(resetValue);}
}
```